



# CSN08x14

## Scripting for Cybersecurity and Networks

### Lecture 4:



# Today's Topics

- Exception (Error) handling
  - *General*
  - *Error-specific*
- Algorithms
  - *What is an algorithm?*
  - *Types of algorithms and common techniques*
  - *Famous algorithms:*
    - Search algorithms
    - Sorting algorithms (in lab)

Go to [www.menti.com](https://www.menti.com)  
code **xxxx**





# Exception handling:

## Dealing with run-time errors in Python





# Python Exceptions (run-time errors)

- **Traceback** is Python showing an unexpected error has occurred at runtime
- Exceptions can and should be intercepted and handled in code

```
>>> f2 = open('non-existent-file')
Traceback (most recent call last):
  File "<pyshell#119>", line 1, in <module>
    f2 = open('non-existent-file')
FileNotFoundError: [Errno 2] No such file or directory: 'non-existent-file'
```

Where Error Occurred

Trace of call stack  
Module/function/lines  
of code which  
generated exception

Error Type

Error description (last thing in trace  
stack)

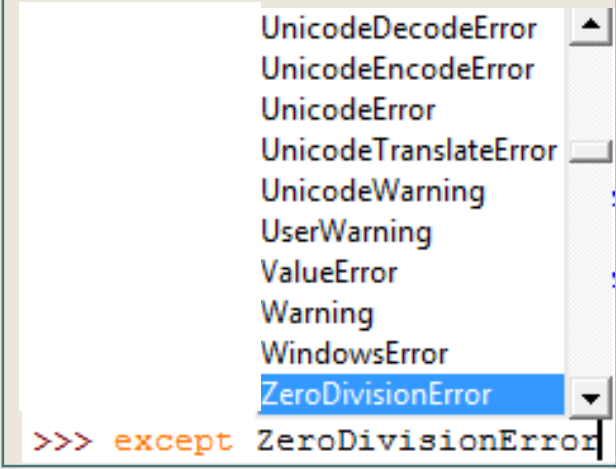


# Common error types

Different exceptions are raised for different reasons. E.g.

- **FileNotFoundError**: trying to open a file that doesn't exist
- **ImportError**: an import fails;
- **IndexError**: a list is indexed with an out-of-range number;
- **NameError**: an unknown variable is used;
- **SyntaxError**: the code can't be parsed properly;
- **TypeError**: a function is called on a value of an inappropriate type;
- **ValueError**: a function is called on a value of the correct type, but with an inappropriate value.

Many error types  
are available

A screenshot of a Python exception list. The list contains the following items: UnicodeDecodeError, UnicodeEncodeError, UnicodeError, UnicodeTranslateError, UnicodeWarning, UserWarning, ValueError, Warning, WindowsError, and ZeroDivisionError. The ZeroDivisionError item is highlighted with a blue background. The list is enclosed in a box with a title "Many error types are available".

UnicodeDecodeError  
UnicodeEncodeError  
UnicodeError  
UnicodeTranslateError  
UnicodeWarning  
UserWarning  
ValueError  
Warning  
WindowsError  
ZeroDivisionError

```
>>> except ZeroDivisionError
```



# Python Exception Handling

- **Try/except** statements are used to catch and deal with exceptions

```
try:
    f=open('non-existent-file')
    for line in f:
        print (line)
    f.close()
except:
    print ('!! Exception: File IO Problem ')
```

Code which may cause Exception - wrapped in try block

If Exception raised, jump to except code

Code to deal with exception in except block  
Code then continues after the try/except blocks



```
>>>
RESTART: F:/Dropbox/CSN08114 Python/SET08115 Python/Lecture4_error_intro.py
!! Exception: File IO Problem
>>>
```



# Handling Specific Error Types

Use the ErrorType and assign as an Exception Object

```
try:
    f=open('non-existent-file')
    for line in f:
        print (line)
    f.close()
except FileNotFoundError as err:
    print (f'!! Exception: {err}')
```

Error Type

Exception  
Object

Specific error  
recovery code



```
----- RESTART: F:/Dropbox/CSN08114 Python/Lecture4_error_part2.py -----
!! Exception: [Errno 2] No such file or directory: 'non-existent-file'
```





# Error-specific Exception Handling

- try/except - specific and general Errors

```
try:
    f=open(r'C:\temp\file.txt')    #c1
    for line in f: print (line)    #c2
    i=5
    print (i+'string')             #c3 # generate value error
    f.close()
except IOError as err:
    print (f'!! Exception: File IO Problem {err}')
except Exception as err:
    print (f'Exception: {err}')
```

**try:**  
Block contains  
code which may  
cause Exceptions

IOError dealt with  
explicitly

**except:**  
Block(s) that  
specify how to  
deal with  
exceptions

All other errors caught  
with general  
**Exception**

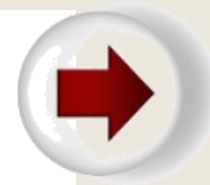
The Error Object is  
assigned to variable  
**err** for more details



# Error-specific Exception Handling

- try/except - specific and general Errors

```
try:
    f=open(r'C:\temp\file.txt')    #c1
    for line in f:    print (line)  #c2
    i=5
    print (i+'string')             #c3 # generate value error
    f.close()
except IOError as err:
    print (f'!! Exception: File IO Problem {err}')
except Exception as err:
    print (f'Exception: {err}')
```



Q: What happens if we have two errors in try block?  
(file doesn't exist and can't add int and string)

Live demo:  
error\_test.py

Q: What is the "r" in open() for??

Q: if #c3 generates error, will file be closed?



# Demo: error\_test.py

error\_test.py (also in moodle)

- What happens if we have two errors in try block?
- only first error will be shown
- Python jumps out of try block as soon as one error occurs

```
try:
    f=open(r'C:\temp\file.txt')    #c1
    for line in f: print (line)    #c2
    i=5
    print (i+'string')              #c3 # generate value error
    f.close()
except IOError as err:
    print (f'!! Exception: File IO Problem {err}')
except Exception as err:
    print (f'Exception: {err}')
```

```
===== RESTART: C:/Python27/error_test.py =====
!! Exception: File IO Problem [Errno 2] No such file or directory: 'C:\\temp\\file.txt'
>>>
===== RESTART: C:/Python27/error_test.py =====
Exception: name 'f' is not defined
>>>
===== RESTART: C:/Python27/error_test.py =====
Exception: unsupported operand type(s) for +: 'int' and 'str'
>>>
===== RESTART: C:/Python27/error_test.py =====
Exception: name 'f' is not defined
>>>
```



# Python Exception Handling: finally

- Run some code if exception or not
- try/except/finally

```
try:
    f=open(r'C:\temp\file.txt')    #c1
    for line in f: print (line)    #c2
    i=5
    print (i+'string')             #c3 |
    f.close()
except IOError as err:
    print (f'!! Exception: File IO Problem {err}')
except Exception as err:
    print (f'Exception: {err}')
finally:
    if 'f' in locals():
        f.close()
```

Code which may cause  
Exception



If Exception raised,  
run  
except code

finally always runs



# Raise an exception explicitly

- Can explicitly raise Exceptions in Interpreter
  - *Good for testing:*
    - No need to think of a test case for each possible exception, and test
  - *Can also be used elsewhere*

```
>>> try:
        raise Exception('My custom message')
except Exception as err:
    print(f'Exception: {err}')
```



```
Exception: My custom message
```

Raises an error with custom msg

Catches and prints custom error



# Algorithms

An introduction





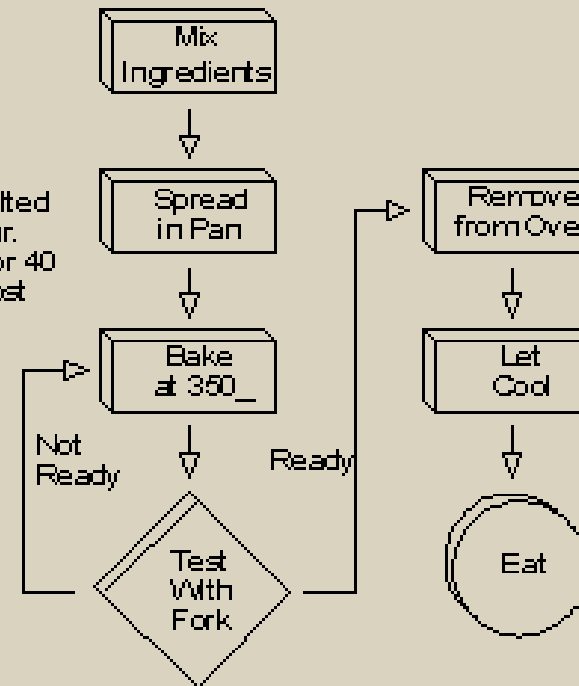
### Recipe CHOCOLATE CAKE

4 oz. chocolate      3 eggs  
1 cup butter        1 tsp. vanilla  
2 cups sugar        1 cup flour

Melt chocolate and butter. Stir sugar into melted chocolate. Stir in eggs and vanilla. Mix in flour. Spread mix in greased pan. Bake at 350\_ for 40 minutes or until inserted fork comes out almost clean. Cool in pan before eating.

#### Program Code

```
Declare variables:  
chocolate  eggs      mix  
butter      vanilla  
sugar       flour  
  
mix = melted ((4*chocolate) + butter)  
mix = stir (mix + (2*sugar))  
mix = stir (mix + (3*eggs) + vanilla)  
mix = mix + flour  
spread (mix)  
While not clean (fork)  
  bake (mix, 350)
```



# What is an algorithm?





# Algorithm - definition

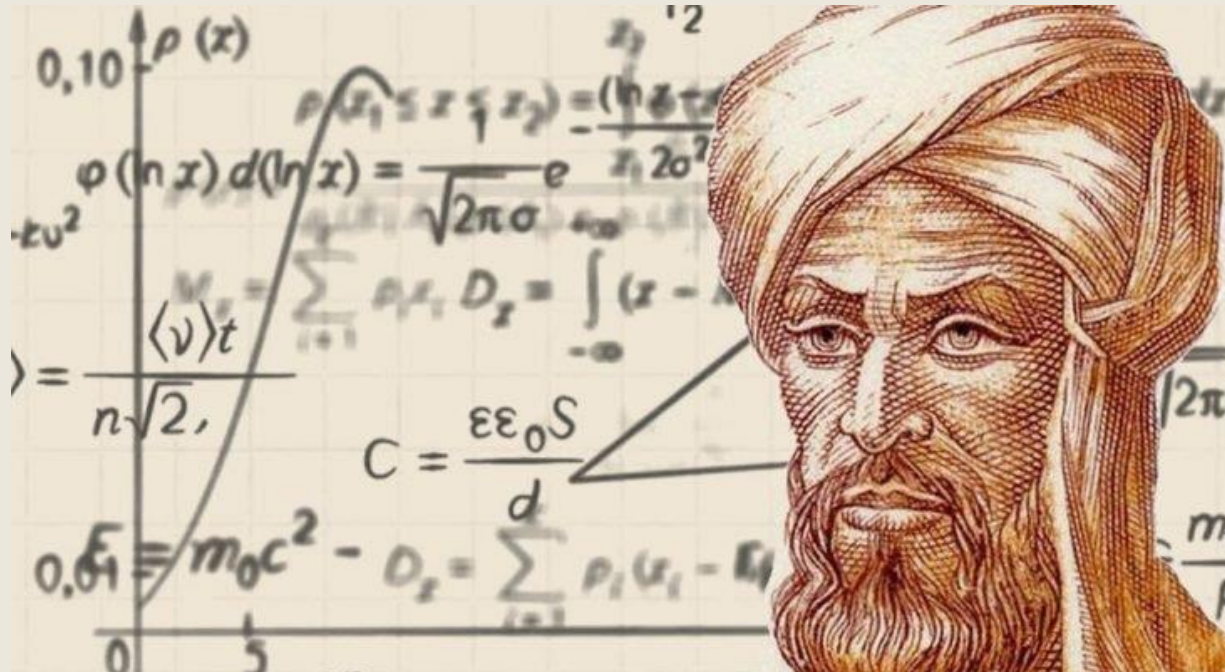
- "a procedure or formula for solving a problem"
- More specifically:
  - “a logical, arithmetical or computational procedure that, if correctly applied, ensures the solution of a problem.”
  - *Must be precise: Sequence of step-by-step instructions*
  - *A set of rules*
  - *Solve a problem in a finite number of steps*
  - *However, in practice not all algorithms always find an optimal solution to the problem*





# Algorithm - origin

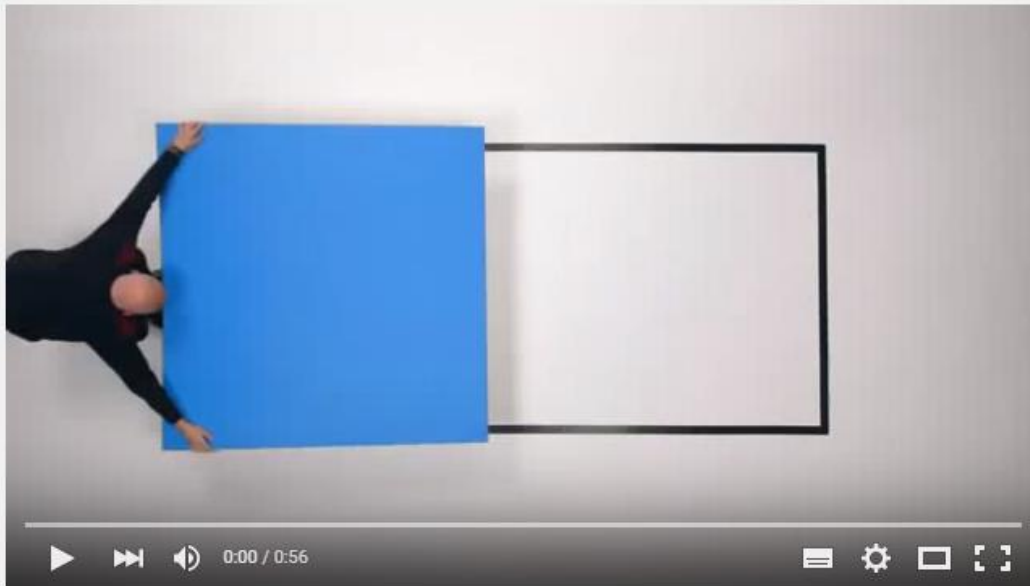
- The word “algorithm” comes from Mohammed ibn-Musa al-Khwarizmi, a mathematician in Baghdad in the 8<sup>th</sup> / 9<sup>th</sup> century.
- Current meaning from 19<sup>th</sup> century





# Marcus Du Sautoy on algorithms (from BBC4)

1 minute intro (now)



Algorithms from "The Secret Rules Of Modern Living: Algorithms"

<https://www.youtube.com/watch?v=UTD2d4KdRKE>

Full 1 hour programme (for later)



Documentary - The Secret Rules of Modern Living Algorithms

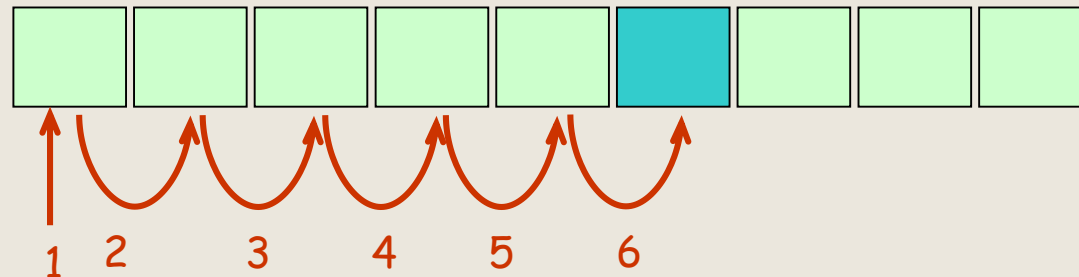
<https://www.youtube.com/watch?v=kiFfp-HAu64>



# Example 1: Linear Search

- start at the beginning of the list we want to search
- compare search value with current value
  - *if search value = current value: STOP*
- repeat until remaining list is empty

## Example





# Linear search in Python

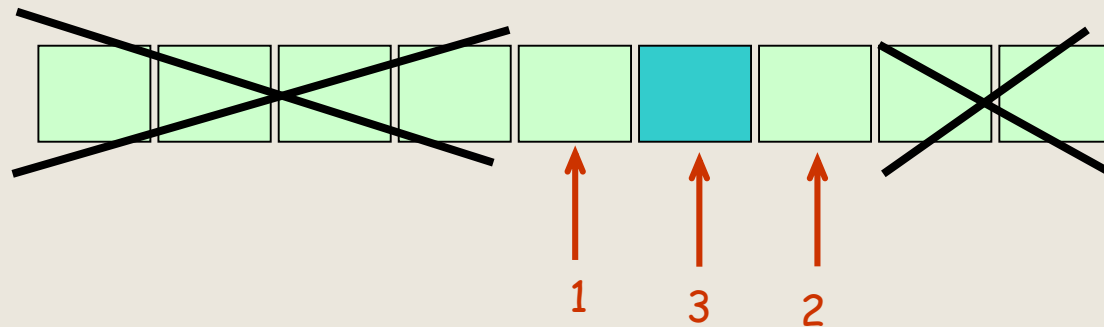
```
def linS(target, words):  
    '''linear search for target in words. words need not be sorted'''  
    for s in words:  
        if s==target:  
            return True  
    return False
```



# Example 2: Binary Search

- compare search value with middle value in list
  - *if search value < middle value:*  
*discard all values > middle (back half)*
  - *Else if search value > middle value:*  
*discard all values < middle (front half)*
  - *Else if search value = middle value: STOP*
- Repeat until remaining list is empty

## Example





# Binary Search in Python

```
def binS(lo,hi,target):  
    '''binary search for target in words.  
    words must be declared elsewhere and a sorted list.'''  
    if (lo>=hi):  
        return False  
    mid = (lo+hi) // 2  
    piv = words[mid]  
    if piv==target:  
        return True  
    if piv<target:  
        return binS(mid+1,hi,target)  
    return binS(lo,mid,target)
```

We will use searching in our case study for this lecture/lab and will come back to compare searching algorithms in the next lecture and lab



# Nested functions in Python





# Binary Search with inner (nested) function

```
def binS(target, words):  
    '''binary search for target in words. words must be a sorted list.'''  
    def chop(lo,hi):  
        if (lo>=hi):  
            return False  
        mid = (lo+hi) // 2  
        piv = words[mid]  
        if piv==target:  
            return True  
        if piv<target:  
            return chop(mid+1,hi)  
        return chop(lo,mid)  
    return chop(0,len(words))
```

- Recursive part is called chop and is defined inside binS
- benefit is that binS will "know" the variables target and words without needing them passed again in every recursive iteration
- See references on next slide





# Nested functions in Python

- <https://realpython.com/blog/python/inner-functions-what-are-they-good-for/> "One main advantage of using this design pattern is that by performing all argument checking in the outer function, you can safely skip error checking altogether in the inner function"
- <http://www.devshed.com/c/a/python/nested-functions-in-python/>
- [https://www.protechtraining.com/content/python\\_fundamentals\\_tutorial-functional\\_programming](https://www.protechtraining.com/content/python_fundamentals_tutorial-functional_programming)
- <https://stackoverflow.com/questions/2005956/how-do-nested-functions-work-in-python>



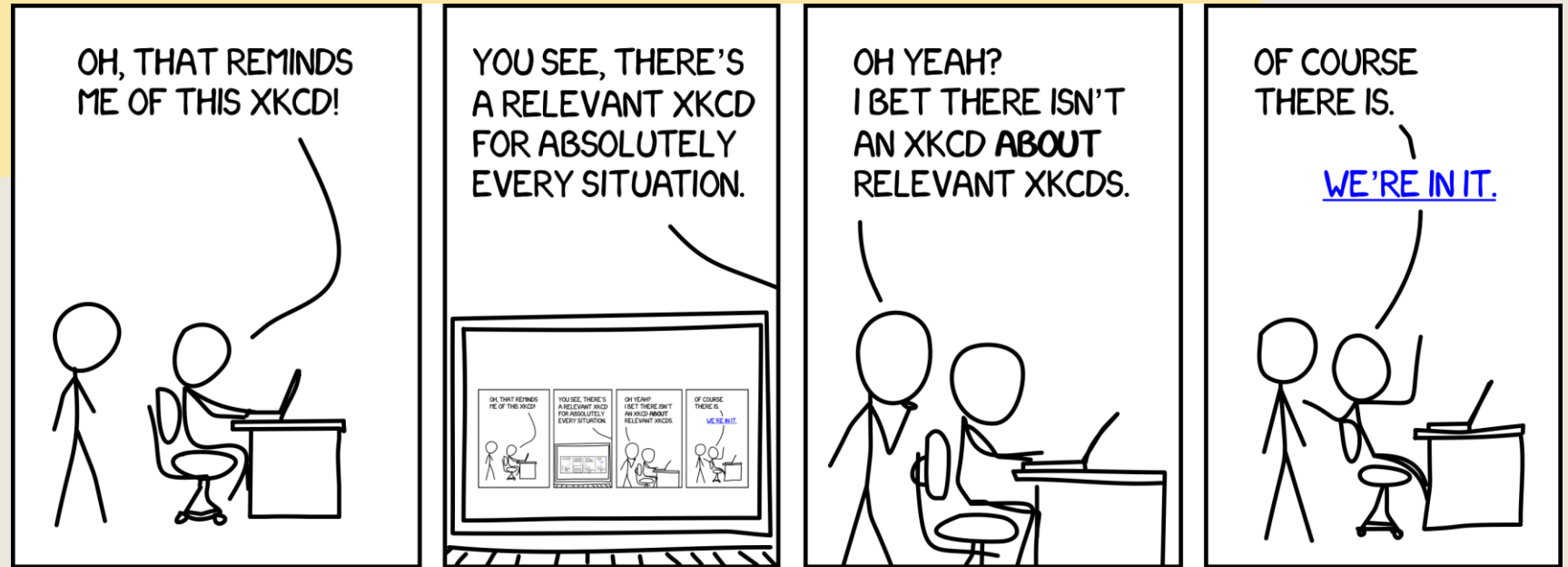
# Types of algorithms and common techniques

- Recursion
- Divide and conquer
- Backtracking (depth first)
- Dynamic programming
- Hill climbing (optimisation)
- Randomisation
- Greedy algorithms





# Recursion



<https://thomaspark.co/wp/wp-content/uploads/2017/01/xkcd.png>

So what is recursion?

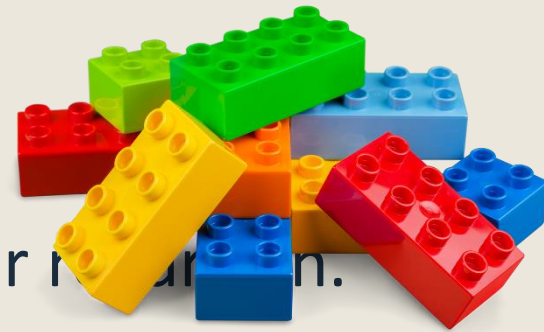
- a method where the solution to a problem is based on solving smaller instances of the same problem.
- An algorithm (or function or program) is recursive if it calls itself one or more times
- See <https://medium.freecodecamp.org/recursion-demystified-99a2105cb871> for explanation and lots of examples



# Building blocks for recursive algorithms

Main requirement is a **Termination condition**:

- with every recursive call the problem is **downsized** and moves towards a base case.
- **base case**: solves the problem without further recursion.
- If the base case is missing (or never met in the calls) – infinite loop
- See [https://www.python-course.eu/python3\\_recursive\\_functions.php](https://www.python-course.eu/python3_recursive_functions.php)





# Examples of recursive algorithms

- Binary search
- Calculating the factorial:  $n! = n * (n-1)!$

```
def factorial(n):  
    if n == 1: # base case  
        return 1  
    else:      # downsizing of problem  
        return n * factorial(n-1)
```

(e.g.  $5! = 5 * 4! = \dots = 5 * 4 * 3 * 2 * 1$ )



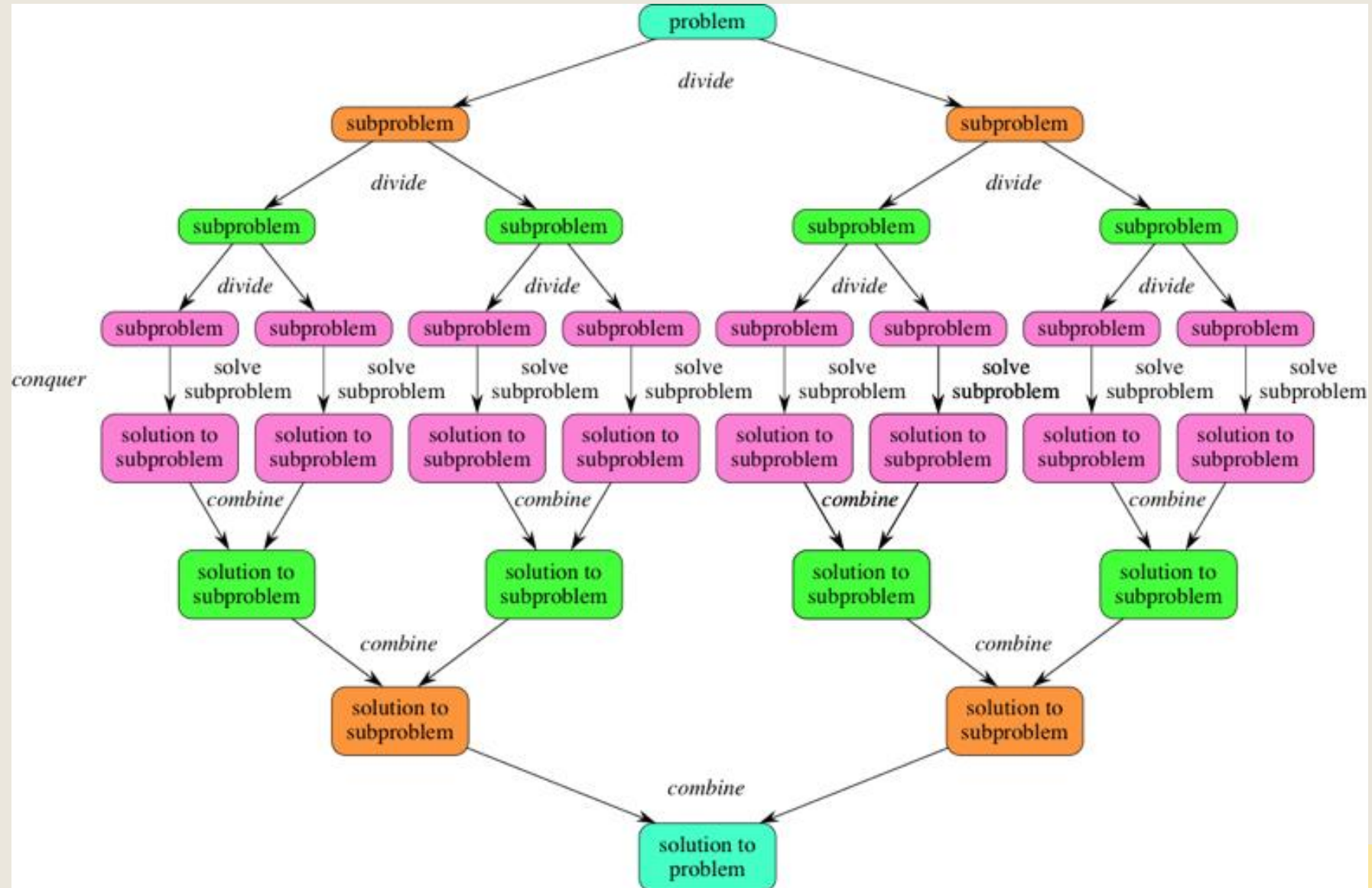
# Divide and Conquer

- Three part process:
  1. **Divide** problem into a set of smaller sub-problems  
(must be instances of the same problem)
  2. **Conquer** the sub-problems by solving them.  
Smallest instances are "base cases" (often solved by brute force)
  3. **Combine** the solutions of the sub-problems into the solution for the original problem (the tricky bit!)
- Steps are recursive
- Examples: binary search, many sorting algorithms

See e.g. <https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms>.



# Divide-and-conquer illustrated



<https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms>





# Example: Calculate $a^b \bmod m$

- The “Obvious” algorithm
  - *Start with 1*
  - *Multiply by  $a$ ,  $b$  times*
  - *Return the value mod  $m$*
- Extremely inefficient
- Useless for larger numbers

```
*exp1.py - F:/Dropbox/CSN08115_MADC/Petra_notes/exp1.py (3.6.1)*
File Edit Format Run Options Window Help
# "Obvious" algorithm
def power(a,b,m):
    acc = 1
    for i in range(b):
        acc = acc * a
    return acc % m

# Some test runs
print (power(3,4,100))
print (power(2,10,1000))
# A prime according to
# https://primes.utm.edu/lists/small/millions/
p = 941083987
# Fermat's Little Theorem says this will be 1
print (power(2,p-1,p))
|
```

Ln: 16 Col: 0





# Example: Calculate $a^b \bmod m$

- The “smarter” algorithm
- Exploit the fact that  $a^{2n+1} = a(a^n)^2$
- therefore

$$a^{2n+1} = a(a^n)^2$$

$$a^{2n} = (a^n)^2$$

...

$$a^0 = 1$$

- This is an example of divide and conquer

```
exp2.py - F:/Dropbox/CSN08115_MADC/Petra_notes/exp2.py (3.6.1)
File Edit Format Run Options Window Help
# Smarter algorithm
def power(a,b,m):
    if b==0:
        return 1
    sqrt = power(a,b//2,m)
    if b % 2 == 0:
        return (sqrt*sqrt) % m
    else:
        return (a*sqrt*sqrt) % m

# Some test runs
print (power(3,4,100))
print (power(2,10,1000))
# A prime according to
# https://primes.utm.edu/lists/small/millions/
p = 941083987
# Fermat's Little Theorem says this will be 1
print (power(2,p-1,p))
Ln: 19 Col: 0
```



# Why calculate $a^b \bmod m$ ?

Let's "park" that question for now



- A **prime** number is an integer that is  $>1$ , and has exactly 2 factors, 1 and itself.
- Very important not only in Cybersecurity
  - *Generating pseudo-random numbers: primes help reduce patterns*
  - *Cryptography: see following slides*
  - See e.g. <https://math.stackexchange.com/questions/43119/real-world-applications-of-prime-numbers>
  - *There are even competitions and prizes for finding new large primes!*



# Prime numbers in cryptography (1)

- Enigma machine: "In order to increase the cipher period of the machine, each wheel has a different number of notches, all being relative primes of 26. Furthermore, there is no common factor between the numbers.  
Wheels I, II and III  
have 17, 15 and 11  
notches respectively."

(<http://www.cryptomuseum.com/crypto/enigma/g/a28.htm>)





# Prime numbers in cryptography (2)

- RSA algorithm
- Public key is product of two very large primes
- Security relies on difficulty of factorising such numbers
- Don't use numbers "too large": "People are used to seeing public keys that are several hundred digits long. If they see one that's several million, ...people are going to start to wonder where you got that prime"  
(<https://blogs.scientificamerican.com/roots-of-unity/psa-do-not-use-the-new-prime-number-for-rsa-encryption/>)



## Mathematics

# Largest prime number discovered - with more than 23m digits

With nearly one million more digits than the previous record holder, the new largest prime number is the 50th rare Mersenne prime ever to be discovered



▲ The new figure (not pictured, sadly) is known as M77232917 and was arrived at by calculating two to the power of 77,232,917 and subtracting one. Photograph: Alamy

Ian Sample *Science editor*

🐦 @iansample

Thu 4 Jan 2018 17.38 GMT

Known simply as M77232917, the figure is arrived at by calculating two to the power of 77,232,917 and subtracting one, leaving a gargantuan string of 23,249,425 digits. The result is nearly one million digits longer than the previous record holder discovered in January 2016.

The number belongs to a rare group of so-called Mersenne prime numbers, named after the 17th century French monk Marin Mersenne. Like any prime number, a Mersenne prime is divisible only by itself and one, but is derived by multiplying twos together over and over before taking away one. The previous record-holding number was the 49th Mersenne prime ever found, making the new one the 50th.

**2<sup>77232917</sup> - 1**

<https://www.theguardian.com/science/2018/jan/04/largest-prime-number-discovered-with-more-than-23m-digits>



How can we check  
whether a given  
number is prime?



# Factorising a number

- If a number is not prime, we can factorise it into a product of prime numbers
- Is 123459 a prime number?
- Let's try to factorise it



- So,  $123459 = 3 * 7 * 5879$ . Therefore it is not a prime



# Proving primality

- X is a prime number if it has no factors
- Try to divide by 2,3,4,5,.....~~X-1~~  $\sqrt{X}$ 
  - *If any of these divisions leaves no remainder, X is not prime, STOP*
  - *If all of these leave a remainder, X is prime*
- Efficiency: we can stop at  $\sqrt{X}$ . Why?
- Efficiency: if we **remember** the primes we have already found we need to divide only by the primes  $\leq \sqrt{X}$





# Pseudo-primes: Fermat's Little Theorem

- Even with efficiency savings, proving primality is still a cumbersome process and takes ages for very large numbers. We need more efficient methods
- **Fermat's Little Theorem**
  - *If  $p$  is prime, then for any integer  $a$ , the number  $a^p - a$  is an integer multiple of  $p$ .*
  - *When  $a$  is not divisible by  $p$ , this means that  $a^{p-1} \bmod p = 1$ .*
  - *$a$  is called a "witness"*
  - *If  $a$  is prime, then  $a$  is not divisible by  $p$ .*
  - *So we can test  $2^{p-1} \bmod p$ ,  $3^{p-1} \bmod p$ ,  $5^{p-1} \bmod p$ , and so on.*



# Using Fermat's little theorem

- There are some composite (non-prime) numbers that also fulfil the equation of the theorem for one or more values of  $a$ .
- Example: **341**=11x31 but  $2^{340} \bmod 341 = 1$
- So if we use Fermat's theorem for testing, we will find some "false positives"
  - *such numbers are quite rare*
  - *The more witnesses we use, the fewer false positives*
  - *But can never rule them out completely*
  - *Carmichael numbers fulfil the equation for all possible witnesses. Smallest is 561 (=3x11x17).*
- Using Fermat's little theorem is therefore a necessary but insufficient test
  - *Numbers that pass such tests are called **pseudo-primes***



# Why calculate $a^b \bmod m$ ?

- Fermat's little Theorem says that  $2^{p-1} \bmod p = 1$  is true for all prime numbers
- So  $a^b \bmod m$  is useful for determining if a number is a prime

```
exp2.py - F:/Dropbox/CSN08115_MADC/Petra_notes/exp2.py (3.6.1)
File Edit Format Run Options Window Help
# Smarter algorithm
def power(a,b,m):
    if b==0:
        return 1
    sqrt = power(a,b//2,m)
    if b % 2 == 0:
        return (sqrt*sqrt) % m
    else:
        return (a*sqrt*sqrt) % m

# Some test runs
print (power(3,4,100))
print (power(2,10,1000))
# A prime according to
# https://primes.utm.edu/lists/small/millions/
p = 941083987
# Fermat's Little Theorem says this will be 1
print (power(2,p-1,p))
|
```



# Backtracking (depth first) algorithms

- Often used for constraint satisfaction problems
- Examples: Sudoku solver; placing  $n$  queens on an  $n \times n$  chess board so that none attacks another
- A clever "trial and error" method
  - *Start with a partial solution vector*
  - *Add next elements to the vector one by one*
  - *If stuck (cannot find an element to add), replace previous element with another possible*
  - *Continue until vector completed*

see <https://en.wikipedia.org/wiki/Backtracking>



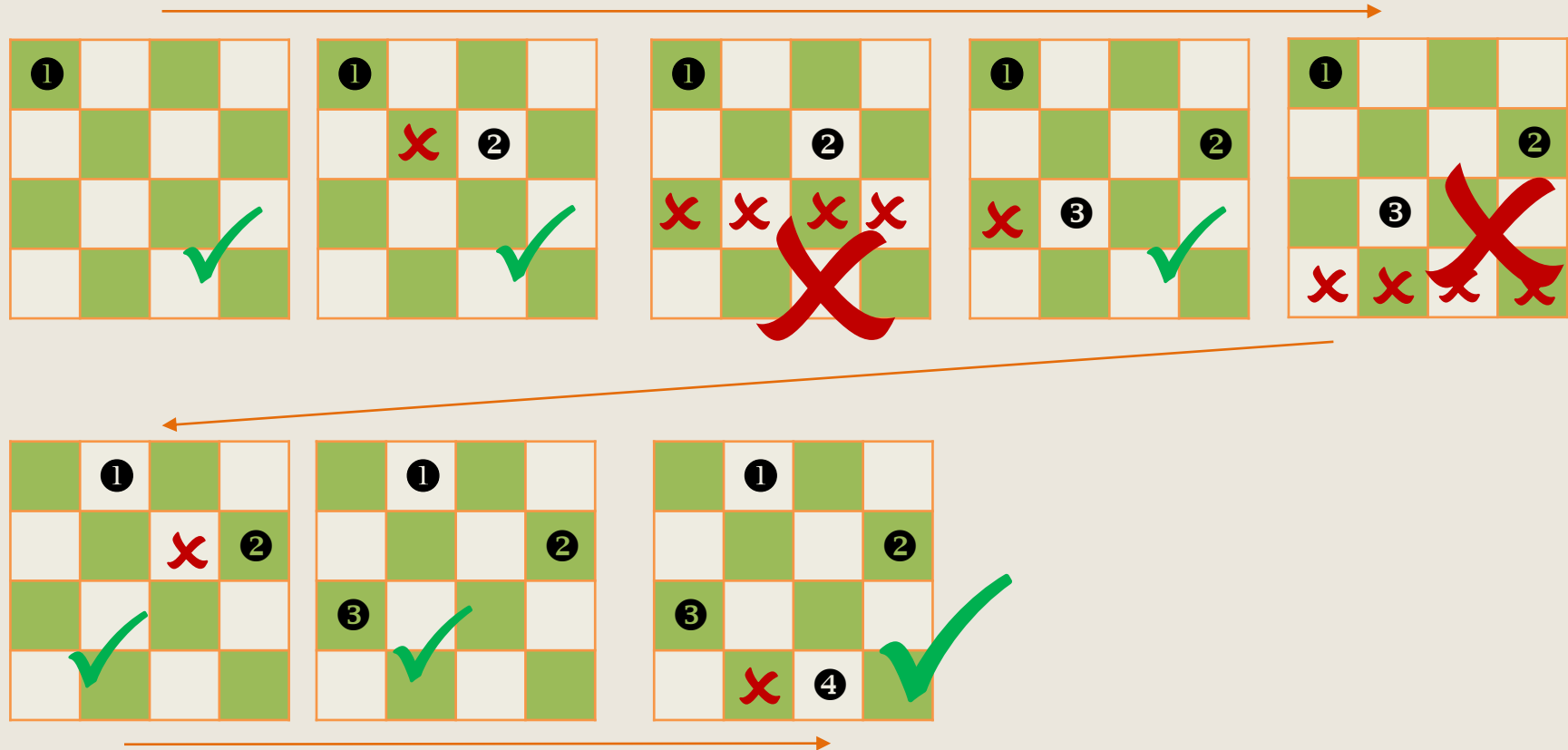
# Backtracking (depth first) example: place 4 Queens on a 4x4 board

Start in first cell in top row.

Place next queen in row below, in next cell along (wrap to left if necessary).

Move to next row if ok, otherwise try next cell along.

If no more cells possible, backtrack: go up to previous row, try next cell along



see <https://en.wikipedia.org/wiki/Backtracking>



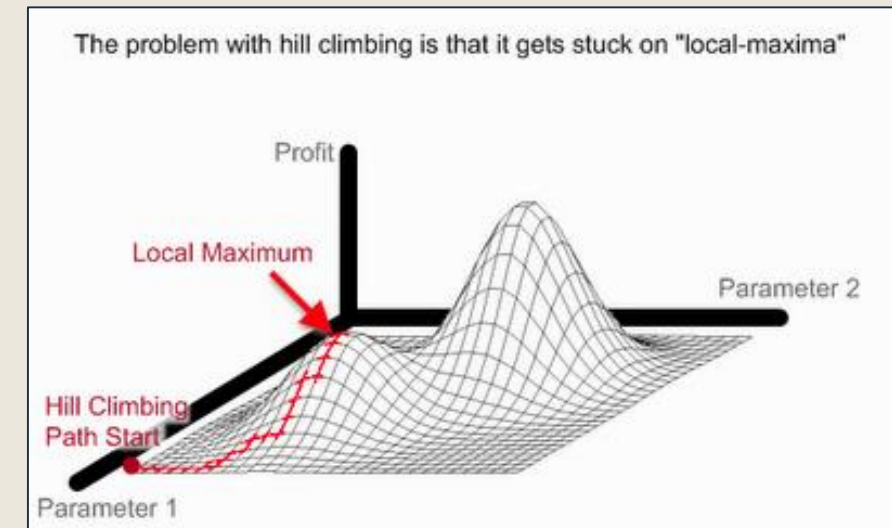
# Dynamic Programming

- Backtracking discards solutions of sub-problems and may need to solve same sub-problem repeatedly
- Dynamic Programming solves all sub-problems first (bottom up – smallest to largest)
  - *stores the solutions in a table*
  - *Each sub-problem visited only once*
- makes backtracking algorithms more efficient



# Hill climbing

- Iterative algorithm of "continuous improvement"
  - *Start with a random (usually poor) solution to a problem*
  - *Change one aspect of the solution; if this improves it, adopt the modified solution*
  - *Repeat until no further improvement can be found*
- Use a "fitness function" to measure goodness of solution
  - *Could be highest altitude, highest profit, highest probability of survival, shortest distance ....*
- Main problem: getting stuck in local optimum  
(try repeating with many different starting points)
- Important example: network flow  
([https://en.wikibooks.org/wiki/Algorithms/Hill\\_Climbing#Network\\_Flow](https://en.wikibooks.org/wiki/Algorithms/Hill_Climbing#Network_Flow))





# Greedy Algorithms

- Make the best choice at each step (according to some criterion)
  - *Instead of considering all sequences and best overall solution*
  - *Like hill climbing that takes steepest ascent at each step*
  - *Not always easy to specify the criterion*
- Only suitable for some problems.
  - *Example: give change, minimising total number of coins – greedy algorithm picks largest possible coin at each step.*

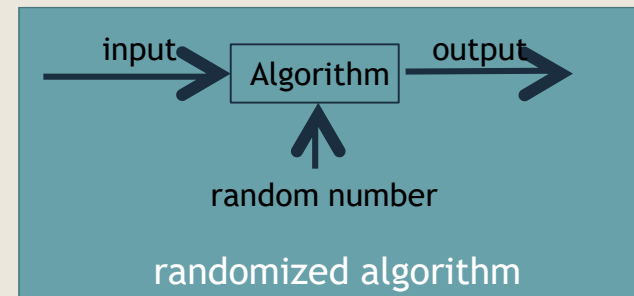
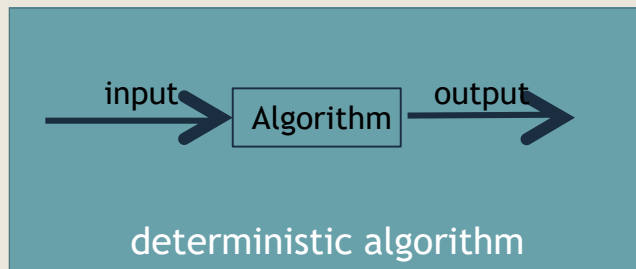






# Randomized algorithms

- Uses a degree of randomness as part of its logic
  - *assign a random value to a variable (according to a fixed distribution, e.g. a coin toss), then use this variable later in the algorithm*
- Possible benefits:
  - *Efficiency: Value of the variable (e.g. outcome of the coin toss) may improve some aspect of the algorithm, e.g. Speed, Space usage.*
  - *Simplicity: algorithm may be conceptually simpler*
- Result can vary if algorithm is run several times with same input
- Example: Quicksort. (chooses pivot randomly). Usually faster than deterministic version.





# Practical Lab 04

Case study:  
Spell checking with Python



# Searching case study: Spell Checker

What do we need?

How is this of any interest in cybersecurity?



This is the first part of the case study.

After the next lecture, we will use it to compare the efficiency of the different approaches



# Three approaches to spell checker

- Linear Search
- Binary Search
- Hash Tables
  - *Python dictionaries are hash tables*
  - *More about this next week*



# Spell check - what you need to do this week

- Write a script to Pre-process the text that is to be checked
  - *Remove special characters*
  - *Write to a new file, one word per line*
  - *This will be the “target” (the text to be spell checked)*
- Write a module that:
  - *Has 2 search functions, linS, binS (each represents one of the algorithms)*
    - You can use the lecture slides for this
  - *Reads the target and a “dictionary” (wordlist) from files*
  - *Checks whether the words in the target are in the dictionary*
  - *Counts & outputs how many words are mis-spelled*



# Spell check – next week

- Write a third function for hash table lookup/search
- Write a wrapper for timing code execution
- Perform efficiency testing - several runs with different sizes of input
- Plot the results
- Consider how to make code even more efficient



# Spell check – pre-processing text to be checked

- Text that is to be checked needs to be pre-processed
- May need to specify encoding as UTF-8 when opening (or decode after)
- Split it into separate words.

```
text = open("shakespeareShort.txt",  
            encoding="utf8")  
content = text.read().split(" ")
```

- Remove quotes, commas, full stops etc.

Hint: have a look at

<https://stackoverflow.com/questions/1059559/split-strings-with-multiple-delimiters>



# Spell check - setting up

- Word lists to be used as "dictionaries" typically have one word per line already
- But need to remove the line feeds when reading from the file
  - Use ***.strip("\n")***
- For binary search, word list must be sorted





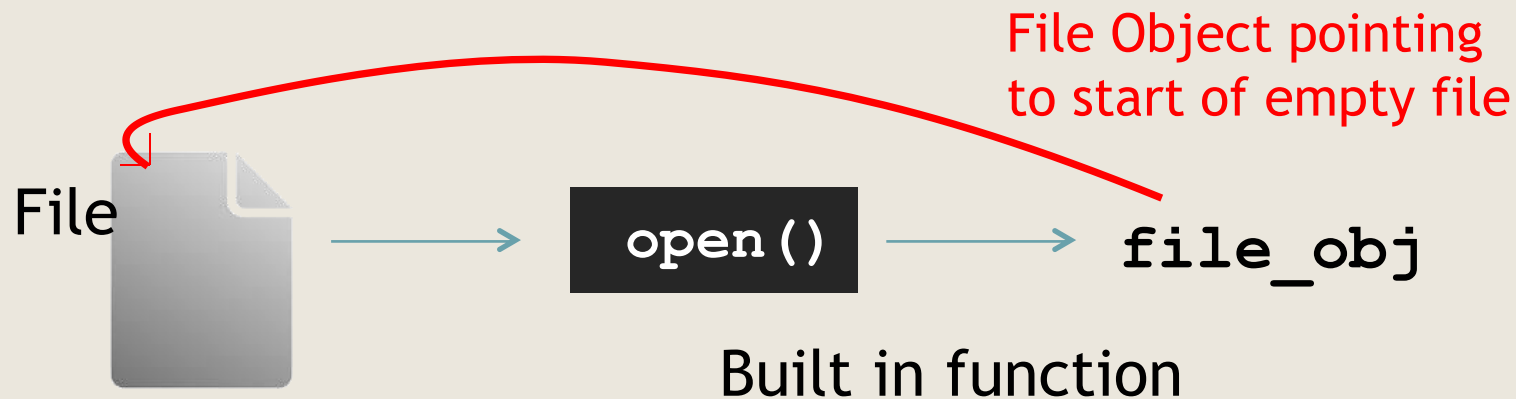
# External Data: Writing to Files



# Writing to File

## ■ 1. Open File in write mode

```
file_obj = open( filename, mode='w' )
```



## ■ 2. Write text to File

```
file_obj.write( 'line of text' )
```

## ■ 3. Close File

```
file_obj.close()
```

Remember that this will encode your text. If you don't specify the encoding, the host OS default will be used (often UTF-8)

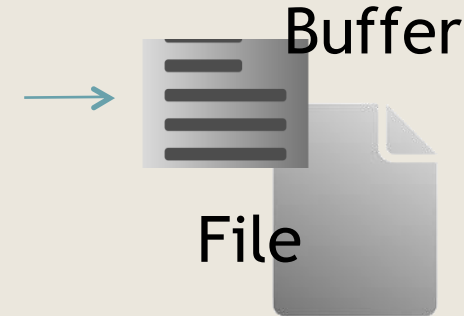


# Writing to File

## ■ Writing Text to File

```
file_obj.write( 'text' )
```

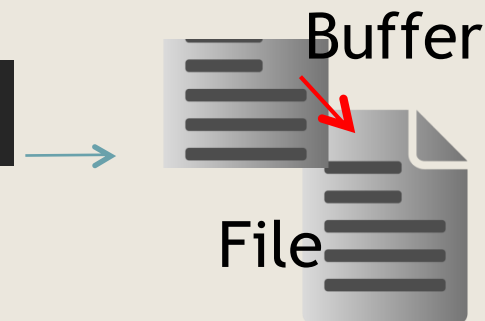
```
file_obj.write('stuff')
```



## ■ Close writes buffer to file and closes

```
file_obj.close()
```

```
file_obj.close()
```



## ■ Flush writes to file while open

```
file_obj.flush()
```



# Writing Text to File

```
file_obj.  
write  
writelines  
xreadlines
```



Str

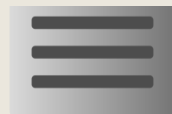


```
file_obj.write()
```



String  
witten  
to file

Sequence



```
file_obj.writelines(seq)
```



Sequence  
written  
to file